A General Model for Authentic Data Publication

Chip Martel*, Glen Nuckolls, Prem Devanbu, Michael Gertz, April Kwong
Department of Computer Science
University of California
Davis, California CA 95616 USA
{nuckolls|martel|gertz|devanbu|kwonga}@cs.ucdavis.edu

Stuart G. Stubblebine
CertCo
55 Broad Street – Suite 22
New York, NY 10004
stubblebine@cs.columbia.edu

Abstract

Query answers from on-line databases can easily be corrupted by hackers or malicious intent by the database publisher. Thus it is important to provide mechanisms which allow clients to trust the results from on-line queries. Authentic publication is a novel scheme which allows *untrusted* publishers to securely answer queries from clients on behalf of trusted off-line data owners. Publishers validate answers using compact, unforgeable *verification objects* ($\mathcal{VO}$s), which clients can check efficiently.

To make authentic publication attractive it is important for the $\mathcal{VO}$s to be small, efficiently computable and verifiable. This has led to the development of a number of data representations for efficient $\mathcal{VO}$ computation. In this paper, we prove the security of $\mathcal{VO}$s for a new general data model called *Search DAGs*. Our security theorem for Search DAGS gives simple security proofs and efficient $\mathcal{VO}$s for a broad range of known structures including binary trees, multi-dimensional range trees, and skip lists. Our approach also helps to provide a clean separation between the proof of security and efficiency.

We also use search DAGs to prove the security of two new and much more complex data models for efficient multi-dimensional range searches. This allows compact $\mathcal{VO}$s to be computed (size $O(\log N + T)$) for typical 1D and 2D range queries, where the query answer is of size T and the database is of size N. We also show I/O efficient schemes to construct the $\mathcal{VO}$s. For a system with disk blocks of size B, we answer 1D and 3-sided range queries and compute the $\mathcal{VO}$s with $O(\log_B N + T/B)$ I/O operations using linear size data structures.

1 Introduction

Large, complex, networked systems often have flaws that allow malicious outsiders to hack into them. Even mature, reliable systems are hard to configure and administer properly. One can rarely be sure that a large information system on the Internet is secure. Thus, the current state of security makes the trustworthiness of online publishing sources suspect.

How, then, can one provide increased assurance for the integrity of high-impact information (*e.g.,* financial, medical, defense) securely on the Internet? In *authentic publication* a *client*, who only trusts a database *owner* (or creator), can use an *untrusted*, third-party publisher for query processing. The *owner* herself can remain safely off-line and simply provide the database periodically to publishers, who answer queries on the owner's behalf. When a *client* submits a query, the *publisher* responds with an answer $\mathcal{A}$ and a *verification object* $\mathcal{VO}$. The client uses the $\mathcal{VO}$ and some digest values provided securely by *owner* to check that the answer is correct (see Figure 1). These schemes provide the following crucial guarantee: if the answer $\mathcal{A}$ to a query is correct the *client* always accepts it, and if $\mathcal{A}$ is incorrect, the *client* will detect that the accompanying $\mathcal{VO}$ is incorrect unless the attacker has found specific collisions in a one-way hash function. We note that in some settings there will be a trusted *certifier* who constructs and securely distributes the digest values.

The use of an untrusted publisher reduces the risks of operating a secure on-line system: an attacker who gains control of a specific *publisher* would not be able to fool *clients*, who would simply find another

*Contact author, phone +1 530 752 3655, fax +1 530 752 4767, email martel@cs.ucdavis.edu. We gratefully acknowledge support from the NSF ITR Program, Grant No. 0085961

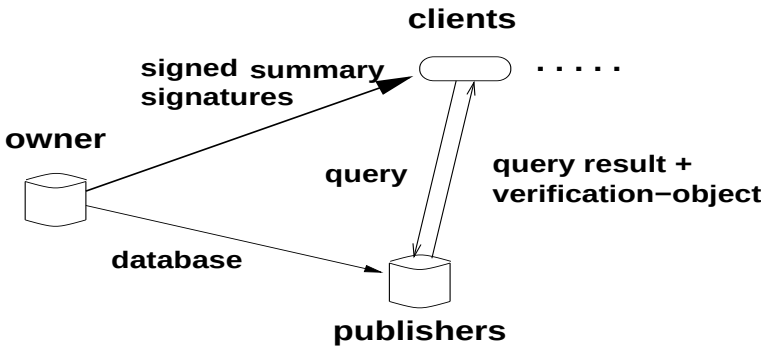

Figure 1: Authentic Publication Scheme

publisher. It also allows graceful scaling by the liberal addition of additional *publishers* in response to increasing demand from *clients*. Note, however, that this approach is currently only practical when the data being queried is relatively static.

$\mathcal{VO}$s must satisfy some critical requirements. First, they must be *secure*: it should be infeasible for a *publisher* to forge an acceptable $\mathcal{VO}$ for a wrong answer. Second, $\mathcal{VO}$s must be *compact*, to reduce transmission overhead. Finally, they must be both *efficiently constructible* by the *publisher*, and *efficiently verifiable* by the *client*. The $\mathcal{VO}$s we construct are cryptographic structures which use one-way hash-functions to make forgery difficult. Previously, work described how to build secure $\mathcal{VO}$s for selection queries, using recursive hash functions over binary trees [6].

The importance of finding good verification schemes for a broad range of queries has already led to the development of secure schemes for a variety of data structures. The original structures focused on membership queries: binary search trees [9], 2-3 trees [10], and more recently, skip-lists [3, 4]. However, we are interested in handling a much richer set of multi-attribute queries such as "return all patches for versions 4.1 to 5.3 of Netscape which were released after July 1, 2000". Solutions for this were proposed using 2D-range trees in [6]. But there are many other data structures which can be useful to support efficient answers to more general types of queries. However, as the data structures get more complex it can be hard to develop authenticated versions of these structures. Thus we introduce a simple yet general data model which we call a *Search DAG* (for Directed Acyclic Graph), and prove a security theorem for $\mathcal{VO}$'s of Search DAGs. Our Search DAG model is general enough to include all the data models listed above and much more (including a very general class of queries). In particular, we can model hybrid data structures (e.g. combinations of trees, arrays, and linked lists), and the use of constraint information associated with the structure (e.g. ranges of values contained in a subtree). We prove that any data model which can be viewed as searching a DAG supports authentic publication and typically does so using $\mathcal{VO}$s whose size and construction time are linear in the search time of the underlying data model. We then use our security theorem to prove the security of two much more complex structures which support efficient solutions to the important setting of multi- dimensional range queries (more details are at the end of this section).

The Search DAG model is intended to deal with the logical view used by the *owner*, *publisher* and *client*. It provides a semantics for digesting information and proving query answers are correct. In essence, we provide a verifiable trace of a correct search for the query answer. The physical implementations used by the publisher to construct $\mathcal{VO}$'s may be quite different (as we will illustrate in section 4.1). Thus the publisher is free to use whatever scheme is convenient without imposing any burden on the client. Our approach also helps to separate the security proof (which deals only with showing the *client* can't be fooled) from the proof of efficiency (size of the $\mathcal{VO}$, time to construct and verify the $\mathcal{VO}$). Finally, our Search DAG model is general enough to allow settings which give the *client* limited information about the answer until the *client* has accepted the $\mathcal{VO}$ (and perhaps now paid the *publisher*).

We now describe several specific new secure data models we develop using our Search DAG model. A

particular focus is on $\mathcal{VO}$'s which can be constructed with good I/O performance since I/O is often the bottleneck for large data sets. To describe our results, we let N be the size of our database, T the size of the answer to a query, and B the size of a disk block used by the query answering system.

We show how to construct $\mathcal{VO}$s efficiently using practical index structures such as B^+-trees, while the logical structure of the $\mathcal{VO}$ itself uses binary trees for simplicity and to allow more compact $\mathcal{VO}$'s. This allows us to answer one-dimensional range queries (e.g. "return all certificates revoked between May 1 and June 1") using a $\mathcal{VO}$ of size $O(\log N + T)$ which can be constructed using only $O(\log_B N + T/B)$ I/O operations.

Our main new results are for multi-dimensional range queries which request all points in a d-dimensional rectangle. [1] This is an important setting since it also models multi-attribute queries: "return all patches for versions 5.1 to 6.3 which were released after July 1, 2000", or "return all credit card numbers revoked between May 1 and June 1 with credit limits between $5,000 and $10,000." In addition, answers to multi-dimensional range queries are important for supporting constraint query languages and queries on class hierarchies in object oriented databases [8]. Answers to 2D range and 3-sided queries (rectangles with one direction going to infinity) are the most important special cases to handle.

Using Search DAGs, we show formally that secure $\mathcal{VO}$s of size $O(\log^d N + T)$ can be constructed for d-dimensional range queries using multi-dimensional range trees (proving the security of the method introduced in [6]) ; we also describe an improved scheme for d-dimensional range queries which has $\mathcal{VO}$ size and construction time $O(\log^{d-1} N + T)$.

For 3-sided range queries we use Search DAGs to convert the data structure described in [1] to get $\mathcal{VO}$s of size $O(\log N + B + T)$ which can be computed using only $O(\log_B N + T/B)$ I/O operations and with linear size data structures. We employ several types of efficient data structures for range queries to compute compact $\mathcal{VO}$s with a small set of security primitives. Additionally, our $\mathcal{VO}$ construction algorithms use the same asymptotic time and space as is needed simply to answer the query. Each of them can be modeled (and proved secure) using our Search DAG model.

Since our Search DAG model makes it fairly easy to create digests and $\mathcal{VO}$s for existing efficient data models, we expect that $\mathcal{VO}$s can be efficiently computed for a wide range of data structures. In particular, the presented schemes provide the basis and guidelines for studying the computation of $\mathcal{VO}$s of more general types of queries in a more formal framework, thus building the foundation of a new area of research on security in information systems.

Related Work. The idea of data authentication has been considered for timestamping [7] and micropayments [13]. The proposed techniques are based on the original work by Merkle [9] and refinements by Naor & Nissim [10] for certificate revocation. Recently, skip lists were shown to be an attractive alternative to tree-based schemes for these settings [3, 4].

These schemes (including ours) share a common theme of leveraging the trust provided by a few signed digest values from a trusted party over multiple hashes, hash paths or hash trees, with the goal of protecting the integrity of the content, with efficient verification, since hashes are more efficient than digital signatures. In [6], the general idea of authentic data publication was introduced and they showed how to securely answer range queries. We extend and improve these results and we provide a general framework for the computation of compact $\mathcal{VO}$s based on different data structures. Necula [11] describes proof-carrying codes, which bundle logical proofs of relevant properties with programs. Note that a similar logic-based approach (proof-carrying answers) in our case would lead in general to impractical proofs the size of the databases themselves. Instead, we use an approach that relies on cryptography.

Structure of the Paper. In Section 2, we introduce the basic security properties of verification objects ($\mathcal{VO}$s) and as an example, show how to efficiently compute compact $\mathcal{VO}$s for 1D range queries. In Section 3, we describe the search DAG model, and prove both its security and efficiency. In Section 4 we show

[1] An example 2D-range query is: return all points (x, y) such that the x coordinate is between 10 and 15 and the y coordinate is between 30 and 50.

that several existing secure structures can be are easily modeled as search DAGS, and we then apply search DAGs to more complex settings. In Section 5, we extend this framework to multi-dimensional range queries, focusing on the general case as well as specialized queries (3-sided queries) and efficient computation schemes (fractional cascading). In Section 6, we summarize our results and outline future work.

2 1-Dimensional Range Queries

In this Section, we outline the principles underlying our data publication scheme and illustrate their use for 1D range queries.

2.1 Basic Scheme

Authentic publication protocols involve three parties: the *owner*, who creates and is responsible for the content; [2] the *publisher*, who handles on-line query processing, and *client*, who submits queries. The *client* relies on the *owner* to create accurate data, but does not trust the *publisher*. The authenticity of our schemes rely primarily on *one-way hash functions* (OWHF). Our protocols typically involve several steps. First, the *owner* computes a digest d of the content using a OWHF over a data structure containing all the data. This d is distributed securely to *clients*, perhaps using a public-key signature scheme. The *owner* then sends the data to the *publisher*. When queries are received from the *client*, the *publisher* sends back an answer $\mathcal{A}$, along with a verification object $\mathcal{VO}$. Using the $\mathcal{VO}$ and $\mathcal{A}$, the *client* can recompute d to verify that $\mathcal{A}$ is exactly what *owner* would have given. In general, we seek to guarantee the following important security property:

Definition 1 An authentic publication protocol involving a *client*, *publisher* and *owner* is **secure** if, given a digest (computed by *owner*), a $\mathcal{VO}$ (computed by *publisher*) and an answer, the *client* will only recompute the digest when the answer is just what the *owner* would have given, unless the *publisher* has engineered a collision in the hash function used.

The actual details of the construction of the digest, $\mathcal{VO}$ etc. vary with the type of query and data model. The rest of this paper focuses on efficiently constructing digests and $\mathcal{VO}$s that are used in secure authentic publication protocols. As background, we recapitulate from [6], the use of binary search trees to authentically publish one-dimensional (1D) range queries over an ordered set of data items $x_1 < \ldots < x_n$ (higher-dimensional queries are discussed later). We build a binary search tree whose leaves are associated with the x_i values, and compute a digest of the tree thus: using a OWHF h, the value of a leaf is the hash of its associated x_i value ($h1 \ldots h4$, as shown in Figure 2). The tree is digested by having each internal node compute the hash of the values of its children. This construction, due to R. Merkle [9], is called a Merkle hash tree and it has been used by several authors to solve authentic publication problems [10, 7, 6]. The root hash value h^* is distributed securely by the *owner* to the *client*. The data is then distributed to the *publisher*, who can build the same binary tree, and recompute the hash values.

To better illustrate our general data model, where we want to be able to represent structural information, we will introduce a slight variant on the standard Merkle tree. Assume that each internal node v_i has a data value d_i associated with it, where d_i is the largest value in the left subtree of vertex v_i (thus when searching for a value x, we go left at node v_i if $x \leq d_i$ otherwise right). The digest value of an internal node v_i will now be $h(d_i, L, R)$ where L, R are the digest values of the left and right children of v_i (see figure 2A). With this type of digest it is easy to show that a value is (or is not) in the tree. For example, to show 50 is not in the Figure 2 tree we give as our $\mathcal{VO}$:

45, $h12$, $h34$
52, $h3$, $h4$
52

[2] there might instead be a *certifier* who is responsible for assuring the correctness of the data and distributing a digest

The first three values hash to the h^* confirming that 45 is the correct split value, so we know to move right (to the node with digest value $h34$). The next three values hash to $h34$ confirming 52 is the correct split value, so we know to move left. Finally 52 hashes to $h3$ (the left child digest value) confirming it is the correct leaf value and that 50 is not in the tree.

In general this digest of the tree allows the *publisher* to give the *client* a $\mathcal{VO}$ which allows the *client* to simulate a proper search in the tree and be convinced that the final leaf values reached are the correct ones. Similarly, suppose the client asks for values in the range (50, 60). The *client* wants to conduct a search for all such values. This search also goes right at the root (since $52 > 45$) and then splits at $h34$ (since $50 \leq 52 < 60$) so both children are explored and the search ends. To demonstrate this search, the *publisher* now returns the above $\mathcal{VO}$ (for the start of the search), plus the value 78. Since 78 hashes to $h4$ we know it is correct. This confirms that 52 is the last in range value, and thus the *client* knows 52 is the correct answer to this range query (what *owner* would have given). The collision resistance of h guarantees that *publisher* could not feasibly forge false values.

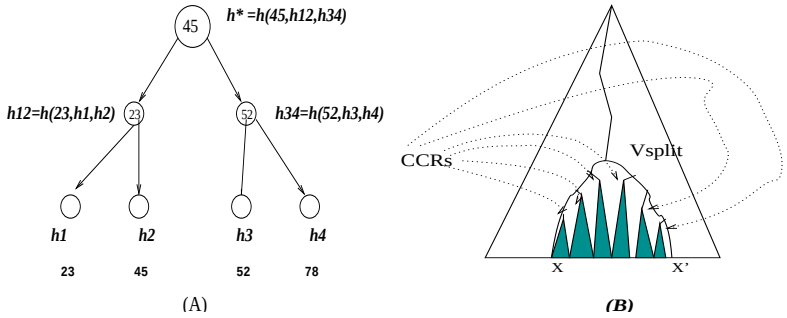

Figure 2: Computation of digest values (A), Split node and canonical covering roots (CCRs) (B)

Figure 2(B) shows how $\mathcal{VO}$s are constructed in an arbitrarily large binary search tree, for a given range query. The split node V_{split} is the lowest single node that covers the entire range; below this node are several canonical covering roots (CCRs) which are roots of balanced binary trees whose leaves together cover the entire answer range (so each is searched entirely). The $\mathcal{VO}$ consists of triples for a search down to V_{split}, the triples for the searches to the left and right extremes (x, x' in Figure 2B), and for the searches within each CCR. This allows the *client* to verify that *publisher* has given the right answer.

We note that this setting is somewhat less efficient (since we hash three values instead of two) but it gives some additional flexibility (which we will use for multi-dimensional searches) and can make the use of the $\mathcal{VO}$ more transparent.

3 A Generalized Model for Query Verification

In the authentic data publication setting the owner determines the most important part of the query verification structure: the logical view which supports digesting and query verification. In this section we introduce a general model for digesting data and verifying answers to queries on the data. It is general enough to capture most existing data structures with search procedures and allows general use of constraint information about the data structures to assure the completeness of answers for a variety of queries.

3.1 Defining the Search DAG Model

A *Search DAG* consists of a directed acyclic graph (DAG) $G = (V, E)$ and an associated search procedure P (we define P more precisely shortly). G has a unique *source* node s with in-degree zero. For each $v \in V$, the owner defines data associated with v which is denoted $a(v)$. This data can contain information about

the successors of v as well as any other information the owner decides is relevant to the search procedure (such as constraint information). The *sink* nodes of the graph have out degree zero and typically contain the real data used in answering queries. We assume that for each non-sink node, its successors are specified in specific order (so we can refer to, e.g., the third successor of v).

The owner is also responsible for defining a search procedure P which takes a query q and searches G to find the appropriate associated data and return the correct answer for q. We assume that P proceeds in the following manner: it starts by reading the query q to be answered. It then searches G by:

(1) Reading the source data $a(s)$
(2) Outputting the next node v_2 to be visited (where v_2 is a successor of s)
(3) Reading $a(v_2)$ then outputting the next node v_3 to be visited (v_3 is a successor of s or v_2)

P continues in this manner (always visiting successors of previously visited nodes) until it completes its search, then outputs **done** and Q, the actual answer to the query q. P can also output **reject** if it reads an invalid or inconsistent data item or a bad query. For convenience of our verification procedure, we will assume that the *next vertex* output of P is in the form (j, k) which means: next visit the kth successor of the jth node visited (e.g. (3,2) says next visit the 2nd successor of the third node visited in the search).

For any query q, the correct answer to q is defined to be the output of P when run on the owner's DAG G. This lets us define our setting for a broad class of queries without having to specify their specific semantics.

We note that in many settings the *client* will know the properties of G (e.g. for a binary search tree) and will be able to do the search without being given an explicit procedure P. Typically the non-sink nodes will have associated data which guides the search and the sink nodes will have associated data from an underlying data set D. An example is the binary search tree tree described in Section 2 where a split value is stored at each internal node of the tree, and the data values are stored at the leaves.

3.2 Digesting the DAG

The owner computes the digest value of the source in G using a one-way hash function h. The digest function f applies to every node in G and has a simple recursive definition:

$$f(v) = \begin{cases} h(a(v)) & : \quad v \quad \text{is a sink node} \\ h(a(v), f(v_1), f(v_2), \ldots, f(v_k)) & : \quad \text{where} \quad v_1, \ldots, v_k \quad \text{are the successors of } v \text{ in order} \end{cases}$$

We can now describe the general authentic publication scheme at a high level. The *owner* chooses an appropriate Search DAG for his data set and gives a copy to *publisher* along with h. Using a secure procedure, the *owner* vsends the *client* the digest value $f(s)$, the "root hash" h and the search procedure P. We also assume that P and h only need to be sent once, while $f(s)$ may need to be resent periodically to reflect updates to the *owner*'s data set. We now show how verification of answers works.

3.3 The Verification Objects and Verification Procedure

We have defined correct answers to queries and how to digest our DAG. We now have to show that an untrusted publisher can provide a $\mathcal{VO}$ for any query defined for the search procedure P. We now describe the structure of such a $\mathcal{VO}$ and the *client*'s verification process. Though other variants on our scheme are possible, we chose a fairly simple verification procedure for clarity, but it will be evident that a fairly large number of implementations are possible which still adhere to the basic model.

We start by describing a correct $\mathcal{VO}$ for a query q, which we will denote $\mathcal{VO}(q)$. Let $v_1(= s), v_2, \ldots, v_n$ be the nodes visited when P is run with input q on the *owner*'s DAG. We also let $u_1^i, \ldots, u_{k_i}^i$ be the successors

of v_i. The $\mathcal{VO}$ for q is then:

$a(s), f(u_1^1)), \ldots, f(u_{k_1}^1)$
$a(v_2), f(u_1^2)), \ldots, f(u_{k_2}^2)$

$\ldots$
$a(v_n), f(u_1^n), \ldots, f(u_{k_n}^n)$

We refer to *lines* of this $\mathcal{VO}$ in a natural way. An example of this type of $\mathcal{VO}$ is given in section two for binary search trees. As in that case, the *client* verifies that the first line is correct by hashing the individual items and comparing the result to $f(s)$. If correct, he inputs $a(s)$ to P. If the next node visited is the jth successor of s, the first output from P will be $(1, j)$. The second line is then checked by hashing it and comparing the result to $f(u_j^1)$, and if it matches, we input $a(v_2)$ to P. The next output is (j, k), (with $j = 1$ or 2). We hash line three and compare the result to $f(u_k^j)$, and input $a(v_3)$ to P, and so on until after inputting $a(v_n)$ P will produce the answer and halt.

In general a correct $\mathcal{VO}$ will always have the following syntax: it consists of a sequence of vectors of integers, with the values in each vector separated by commas, and each vector is terminated by a newline (we could of course use any consistent delimiters). Any $\mathcal{VO}$ not in this form is immediately rejected. Thus any syntactically correct $\mathcal{VO}$ can be described as:

$x_1, y_1^1, y_2^1, \ldots y_{k_1}^1$
$x_2, y_1^2, y_2^2, \ldots y_{k_2}^2$
$\ldots$
$x_n, y_1^n, y_2^n, \ldots y_{k_n}^n$

and we use $\overline{l_i}$ to refer to the vector of values in the ith line of the $\mathcal{VO}$.

Given a query q and a syntactically correct $\mathcal{VO}$ V the verification process, $V_P(q)$, for V has a quite simple structure. It starts by inputting q to P. Then the verification process is:

step 1 **:** Hash the values in $\overline{l_1}$ and compare the result to the value $f(s)$. If they match, input x_1 to P and get output $(1, j)$. If they mismatch, reject.

step 2 **:** Hash the values in $\overline{l_2}$ and compare to y_j^1. If they match, input x_2 to P and get output (i, j_2). If they mismatch, reject.

step $i = 3$ **:** Hash the values in $\overline{l_3}$ and compare to $y_{j_2}^i$. If they match, input x_3 to P and get output (i_3, j_3). If they mismatch, reject.

step $i = 4$ **and up :** Continue in this manner until P halts and the answer is output (the verification is successful), a computed value mismatches (reject V), or you run out of lines (again reject V).

In essence, V_P will proceed just as P would when searching G except that, at each node, the additional verification data for that node is processed. It is again up to the *publisher* to make sure that the sequence of nodes is correct. However, a mistake by the *publisher* will just cause the *client* to reject the $\mathcal{VO}$.

3.4 Security Theorem for the Verification Procedure

Here we give a result which proves that a $\mathcal{VO}$ is accepted by our verification process V_P only if $V_P(q)$ verifies and extracts the same query data that the owner would have, unless the publisher was able to forge the $\mathcal{VO}$ for q. We first consider a particular type of bad $\mathcal{VO}$ which could trick our verification procedure.

Definition 2 A syntactically correct $\mathcal{VO}$ V is a **forgery** of $\mathcal{VO}(q)$ if V has a line $\overline{l_i}$, for $i \leq N$, the number of lines in $\mathcal{VO}(q)$, such that $\overline{l_i}$ is not the same as the ith line of $\mathcal{VO}(q)$, but both lines hash to the same value using h.

We note that forgery is a necessary condition for fooling our verification, but it is often not sufficient. For example, even if an attacker uses an alternate vector $\tilde{x}_i, \tilde{y}^i_1, \tilde{y}^i_2, \ldots \tilde{y}^i_{k_i}$ which has the same hash value as the correct vector $x_i, y^i_1, y^i_2, \ldots y^i_{k_i}$, the value $\tilde{x}_i$ may well not be a valid input to P, so will be rejected. We get the following theorem directly from our definition of a correct $\mathcal{VO}$ and query answer.

Theorem 3 If the user is provided with a correct $\mathcal{VO}$ for a query q, then $V_P(q)$ will accept it, and return the correct query data set Q.

Note that if $\mathcal{VO}(q)$ has N lines, we will accept a $\mathcal{VO}$ V as long as its first N lines match $\mathcal{VO}(q)$ even if it has additional lines.

Theorem 4 Given a candidate $\mathcal{VO}$ V, if $\mathcal{VO}(q)$ is not a prefix of V and V is not a forgery of $\mathcal{VO}(q)$, then $V_P(q)$ rejects V.

proof: We prove that for all n up the length of $\mathcal{VO}(q)$, if the first $n-1$ lines of V are correct, but line n is incorrect, then $V_P(q)$ rejects V after processing line n. The theorem follows immediately from this. The proof is by induction on n.

If $n = 1$,
$V_P(q)$ starts by hashing line one of V and comparing this to $f(s)$. Since V is not a forgery of $\mathcal{VO}(q)$ these two values match only if line one is the correct vector.

induction step
Now assume that the first $n-1$ lines of $V_P(q)$ are correct (with $n \geq 2$). We show that $V_P(q)$ rejects after processing line n unless $\overline{l_n}$ is correct.

Since the first $n-1$ lines are correct, P has been given the correct first $n-1$ data values, and thus its output (which we denote (j,k)) after step $n-1$ is the correct next node to visit. $V_P(q)$ will next compare the hash of line n with y^j_k. Since $j < n$ we know y^j_k is the correct value to compare to (again by the induction hypothesis), and since V is not a forgery of $\mathcal{VO}(q)$, the hash of line n will only match if the vector of values is the same as in line n of $\mathcal{VO}(q)$. □

This shows how to securely publish data that can be represented as a Search DAG. However, so far it does not say anything about efficiency. This will of course depend on the DAG G and the search procedure P. In fact, for any data set D and query q, we have the trivial search procedure P which looks at every data item in D and then answers q. Our focus is of course on Search DAGs created from efficient search procedures, and we can prove an efficiency theorem for an important class of Search DAGs. To do so let $N(q,D)$ be the number of nodes visited in the search which answers q and $T(q)$ be the time taken by P to process q before it starts the search.

Theorem 5 Consider a Search DAG where G has bounded degree, the data values $a(v)$ associated with the nodes are of bounded size, and P can process a data value $a(v)$ in $O(1)$ time. Then for any query q, we can build $\mathcal{VO}$s of size $O(N(q,D))$ which can be constructed and verified in time $O(N(q,D) + T(q))$.

proof: The $\mathcal{VO}$ for q has $N(q,D)$ lines each of $O(1)$ size by the bounded assumptions of $a(v)$ and the degree of G. If each line is processed in $O(1)$ time the verification/construction time follows. □

We note the above boundedness assumptions apply to many normal search procedures such as binary search trees, multi-dimensional range trees, and skip-lists. In the next section we show that it is easy to cast these

in our model and thus get good $\mathcal{VO}$ schemes. However, the main advantages of our model is for the more complex data structures we deal with in the following sections.

We also note that verifying by simulating the search in G provides a general method, but may lead to some inefficiencies. However, once the basic digesting/verifying method is known, it is often easy to modify the verification process to exploit specific properties and improve efficiency. Most common is to do the verification "bottom-up" by starting with the values from the fringe of the search and hashing them to get the predecessor node's value. We will also see that we can sometimes transform nodes with high degree into a tree of lower degree nodes, thus reducing the size of our $\mathcal{VO}$.

4 Efficient construction of $\mathcal{VO}$'s for Dictionaries

In this section, we discuss several ways to build $\mathcal{VO}$s for structures which support dictionary queries: is element x in the data set? The structures also can support efficient insertions and deletions of new elements. We argue that Search DAGs give easy security proofs for several standard structures.

The binary Merkle tree discussed in Section 2 is the classic way to support an authenticated dictionary and it also defines a compact $\mathcal{VO}$ for a 1D range query.

As indicated in Section 2, if we use as a Search DAG the binary tree with the split value and whether the node has a left or right child as its associated data, a simple search procedure results in a Search DAG. It is also trivial to convert a 2-3 tree into a Search DAG if one wants to support efficient updates (as in [10]).

Skip lists [12] can provide an attractive alternative to trees, and Goodrich and Tamassia recently showed how to create efficient $\mathcal{VO}$s for skip list answers [3, 4]: $O(\log n)$ size with small constants. A skip list is easily viewed as a DAG, and their digesting process mirrors what could be done in a natural Search DAG for skip lists: each node has its associated value and whether it has a down and/or right pointer. Their figure 5 describing the digesting process would in fact be a good figure for the associated Search DAG if the arrows were reversed [3]. Of course, they do get better constants by using a bottom up approach and introducing a commutative hash function. But the basic digesting structure is fairly easy to find using the Search DAG approach. The only trick is splitting the nodes at the bottom level. This isn't necessary for search, but makes updates faster.

4.1 I/O Efficient construction of $\mathcal{VO}$'s

The prior data structures are all good for main memory implementations, but can have poor I/O performance. This is particularly important when querying large data sets. The classic way to get good I/O performance is to use a B-tree. Again, it is easy to think of a B-tree in Search DAG terms where each node has the $B-1$ split values to decide where to go next. Unfortunately, this would lead to larger $\mathcal{VO}$'s of size $B \log_B N$ for data sets of size N. We now show that a better solution is to have the *owner* use a binary tree as the *logical* representation of the $\mathcal{VO}$, but have the *publisher* use a B-tree as the physical representation to construct the $\mathcal{VO}$. This emphasizes the fact that the Search DAG is only a logical view of the data, and need not restrict the publisher's physical implementation. This also demonstrates a general method which will often allow high degree Search DAG nodes to be replaced by a tree of lower degree nodes.

To describe our setting, assume N is the number of values in our data set, B the size of a disk block and T the number of values in our answer. We also use $n = N/B$ and $t = T/B$. Let MT denote the binary tree which is the Search DAG graph built by the owner.

We use a multi-way tree BT with branching factor $\Theta(B)$ (such as a B-tree) to simultaneously find the answer and determine the information needed to build the $\mathcal{VO}$. We just pack the binary tree into BT: the root of BT represents the top $\log_2 B$ levels of MT. For each node v in MT, the associated BT node contains the (splitting) key value in v and the hash value associated with v in MT. It is then simple for the publisher to construct the answer and $\mathcal{VO}$ by following the path in BT which matches the path taken in MT to the

split node, and then to follow paths corresponding to those in MT down to the upper limit and lower limit leaves (x and x' in Figure 2(B)). Along the way, the needed hash values for the $\mathcal{VO}$ will be either in the BT nodes on the paths used or in a sibling node. Thus we have:

Theorem 6 The (binary) $\mathcal{VO}$ for a 1D range query can be constructed with $\Theta(\log_B N + t)$ I/O operations using a multi-way tree of size $\Theta(n)$ disk blocks.

Proof sketch: Each node in BT can store $O(B)$ split and hash values using O(1) disk blocks. The search looks at O(1) disk blocks at each level of BT for the search paths, and it only looks at two leaf disk blocks (first and last) which do not contain $\Theta(B)$ answer points. □

It is important to note that there is *no need* to show a distinct security property in this case; the *publisher* constructs the same $\mathcal{VO}$ as with the binary tree; she just does this more efficiently.

While the above describes a static multi-way tree, we note that if the owner were to insert/delete items which caused only local changes to MT, we could also modify a B-tree implementation of BT using local changes as well.

5 Multi-dimensional Range Queries

In this section, we introduce efficient computation schemes for multi-dimensional range queries. The general case using range trees is outlined in Section 5.1. In Section 5.2, we discuss an improved computation scheme using fractional cascading. $\mathcal{VO}$s for 3-sided queries, a special and important type of range queries, are detailed in Section 5.3.

5.1 Multidimensional Range Trees

For a database of N points, rectangular queries of the pattern $(x_{l_1}, x_{u_1}), (x_{l_2}, x_{u_2}), \ldots, (x_{l_d}, x_{u_d})$, can be efficiently processed by so-called *multidimensional range trees (mdrts)* [2] in time $O(\log^d N + T)$. In this section, we show that mdrts can be used to efficiently compute $\mathcal{VO}$s for rectangular queries as well. In the following, we first outline the basic characteristics of mdrts and then show how to convert them to a Search DAG to get compact $\mathcal{VO}$s. We will base our discussions on 2D queries. The presented framework easily extends to higher dimensions. The result on mdrts largely duplicates what was done in [6], but we repeat the details since we build on them later in our improved results. This is also the first full security proof for mdrts.

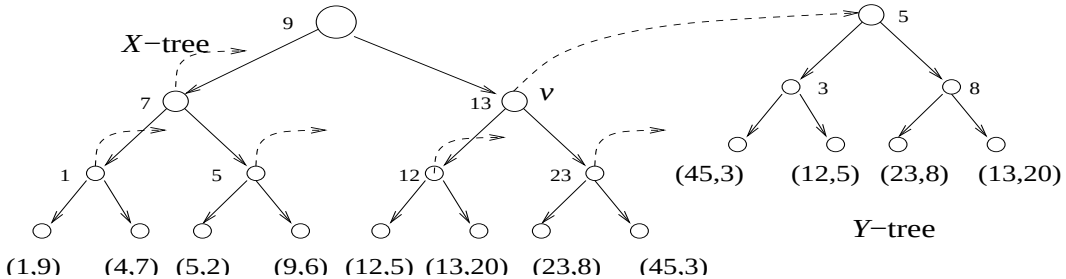

Figure 3: Example of a 2D Range Tree

Consider the example *mdrt* shown in (Figure 3), consisting of one X-tree, and multiple Y-trees which represents points in the 2D space. The X-tree simply sorts the points according to the x coordinate. Each interior node in the X-tree is an ancestor of a set of points represented by a Y-tree. Consider the interior node v in the X-tree, which is the ancestor of points with coordinates (12,5), (13,20), (23,8) and (45,3). An mdrt now contains a link from v to the root of an *associated Y-tree*, denoted as $Y(v)$. This Y-tree contains the same set of points (12,5), (13,20), (23,8) and (45,3); however, in this tree, they are sorted by the y

coordinate. Likewise each interior node v_i in the X-tree is the ancestor to a set of points, and contains a pointer to an associated 2D *mdrt* $Y(v_i)$ which sorts the points in the subtree below v_i by coordinate y. In general, for higher dimension *mdrt*'s, each node v of a $j+1$-dimensional *mdrt* contains a pointer to a j-dimensional *mdrt*. The nodes of the final 1D-tree (Y-tree(s) above) do not have such pointers.

Consider a 2D rectangular query of the form $(x_l, x_u), (y_l, y_u)$. To answer this query, we first (as was illustrated in Figure 2(B)) compute the set of CCRs in the X-dimension; as argued in [2] (pp. 103-107) there are exactly $O(\log N)$ of these, which can be found in $O(\log N)$ time. For each of these, the corresponding Y-tree is searched for the requisite range, thus yielding an $O(\log^2 N)$ time for 2D range searches. In general, it is shown in [2] that d-dimensional range queries can be computed in time $O(\log^d N + T)$. Range trees require $O(n \log^{d-1} N)$ storage space, and can be constructed in time $O(N \log^{d-1} N)$.

It should be fairly clear how to convert an mdrt to a Search DAG: the nodes and arcs of the DAG are exactly as in the mdrt; each internal node has its split value (in the appropriate dimension as in Figure 3) and a tag as its associated values (the tag indicates whether it is a leaf); a leaf has a tag and the (x, y) coordinates of its associated point. This is sufficient to guide a search which finds the CCR roots and then searches all the appropriate leaves in the associated Y trees. A trace of this search is the $\mathcal{VO}$. Each node has degree at most three and a bounded amount of data, so by Theorem five we immediately get a $\mathcal{VO}$ of size and construction time equal to the number of nodes visited: $O(\log^2 N + T)$.

The above 2D construction extends easily to a d-dimension *mdrt* (e.g. for 3-dimensions, each node of the Y-tree points to a Z-tree sorted on the z-coordinate).

Theorem 7 We can verify a d-dimensional range query using a $\mathcal{VO}$ of size $O(\log^d N + T)$ which can be computed in the same time.

Proof sketch: As indicated before, it is easy to convert a d-dimensional mdrt to a bounded degree search DAG, and to then certify a trace of the search for the answer points.

5.2 Improved Multi-dimensional Range Queries

We now show how our Search DAG verification model can be applied to a more efficient multi-dimensional range query structure to produce an equally efficient $\mathcal{VO}$ for queries on the structure. *Fractional cascading* [2, 5] reduces the search time for a d dimensional query by a factor of $O(logn)$ from $O(log^d n + T)$ using a mdrt, to $O(log^{d-1} n + T)$. For brevity, we just describe 2D range queries over sets of points on the plane[3], however, the technique also applies to higher dimensions. We briefly review fractional cascading and use the Search DAG model to construct $O(\log n)$ size $\mathcal{VO}$s for 2D range queries.

With 2D mdrts, we first find $O(\log n)$ X-dimension CCRs and then, for each CCR, we search the associated Y-tree. Each Y-search produces an $O(\log n)$ piece of the final $\mathcal{VO}$. In fractional cascading each node v in the X-tree has a pointer to an associated array denoted $Y(v)$ containing all points in the subtree rooted at v, sorted by their Y-coordinate. Let $Y(v)[i].y$ denote the y coordinate of the ith element of $Y(v)$. Let w, a be the left/right children of v in the X-tree. An element $Y(v)[j]$ has a pointer to $Y(w)[i]$, where i is the smallest index such that $Y(w)[i].y \geq Y(v)[j].y$. $Y(v)[j]$ also has a pointer to the analogous entry $Y(a)[m]$ (see Figure 4).

For a 2D query $(x_l, x_u), (y_l, y_u)$, we first do a 1D range search for (x_l, x_u) in the X-tree, to find V_{split} and the CCR roots as in Figure 4. If V_{split} is a leaf, it is the answer. Otherwise let L and R be the children of V_{split}. $Y(L)$ now has all the points in the subtree rooted at L; we seek the smallest j such that $Y(L)[j].y \geq y_l$. The pointers associated with $Y(L)[j]$ directly indicate candidate Y-range points; we just follow a pointer to the first possible answer element and then scan till we hit an out-of-range point. The cascading pointers allow scans rather than searches, and provide the additional efficiency over mdrts. The search in the X-tree tells us which nodes v under L are roots of CCR's, so we do these scans on the corresponding $Y(v)$ arrays.

[3] We can associate each point with a data value representing an associated tuple, or hash thereof.

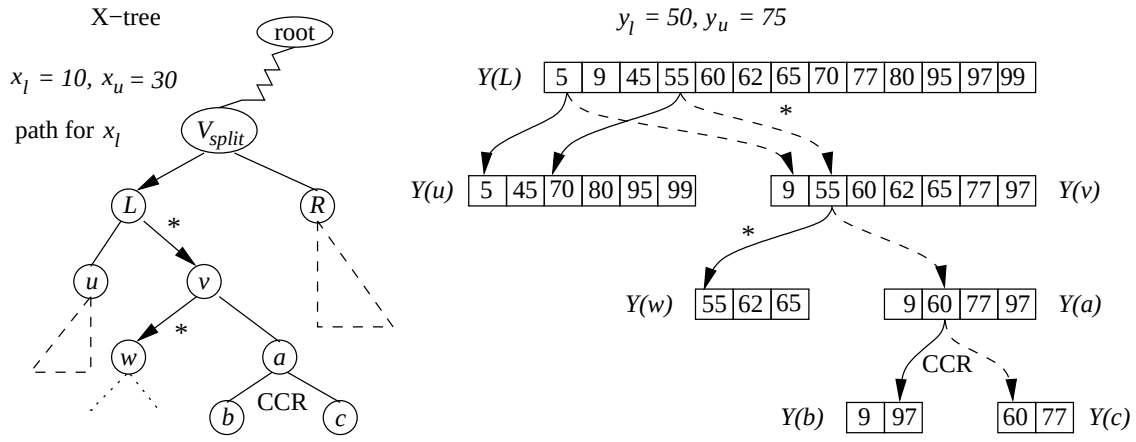

Figure 4: Fractional Cascading Scheme (only y-coordinates and select pointers are shown with the arrays)

The Search DAG Now we show how this search procedure helps determine the entire Search DAG. Our DAG builds on the mdrt Search DAG of the previous section. The X-tree and associated Y-tree's are the same, except that the leaves of a Y-tree now only hold y coordinates and are no longer sinks. These nodes, which were leaves of a tree $Y(v)$ in the regular MDRT, will now have pointers to the corresponding elements of the fractional cascading array. Just like the mdrt, a fractional cascading search will identify the CCR's in the X tree and the split node v_{split}. A search in the Y tree associated with v_{split} lets us find the right place to begin our fractional cascade search (value 55 in the prior example). The search DAG for the "array searches" will closely correspond to the original array structure with a small variation. By tracing though part of a typical search, we can see the motivation for this difference.

Once the smallest y value at least as large as y_l is found, we begin to traverse the arrays corresponding to nodes in the X-tree which are CCRs. At each array corresponding to a CCR, we search to the right until we encounter a y-value outside of the query range. Although the physical implementation of the fractional cascading algorithm might suggest that the nodes at each array have three successors: one to the element to its right and two to its "child" arrays, the logical view indicates that this is not quite the case. When searching the array to the right, the pointers to the lower arrays are ignored. Thus, for the purposes of a sequential array search, there is exactly one successor for each node other than for the right end sink node. This tells us that in the Search DAG, the nodes for the elements of the arrays as used for a sequential search should be distinct from the nodes reached via pointers from higher arrays [4]. This suggests simply keeping two arrays, each supporting one of the two functions of the array in the search (see figure 5). We refer to the array with data used for the sequential search as $Y_S(v)$ and the array with the child pointers as $Y_T(v)$, where v is the associated node in the X tree. We note that the array elements here are actually just nodes in the Search DAG. The associated data field at nodes of $Y_T(v)$ will be empty in our case since the search procedure used the information from the X tree search to determine which of the successor arrays to follow when a node of $Y_T(v)$ is reached and the Y_S arrays are always searched. Finally, to convert this to a DAG, each array element becomes a node. Y_S nodes have data values and one outgoing pointer to its right neighbor. Y_T nodes have no data, only pointers to their two children and to their corresponding Y_S node.

The search in these structures proceeds just as in normal fractional cascading: search the X-tree to find v_{split} and CCRs, go to v_{split}'s Y tree to find the right position to start the fractional cascade, then follow pointers to find the starting point for a linear search of answer points for each CCR.

[4] We could use a single array, but this would produce a different DAG and more complicated digest function.

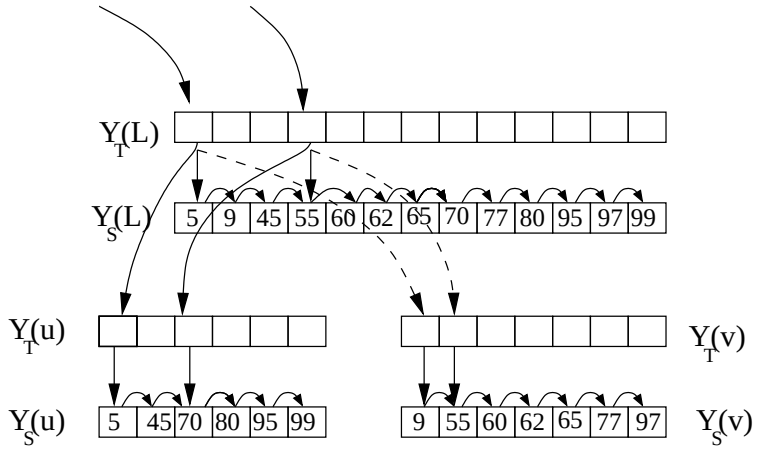

Figure 5: Logical View of the Arrays (only y-coordinates and select pointers are shown with the arrays)

Digesting Fractional Cascades

Although our model tells us that the digest values are simply computed from the sinks of our search DAG up to the source, we now give a simple description of the function for this particular search DAG.

The structure of each Y_S array tells us to hash values right-to-left. Once each of the values for Y_S are computed, the Y_T array values are just the hash of the values of their three successors. Each entire Y_T array is digested using the binary Y-tree associated with it, and then the X-tree is digested in the usual way with the addition of the digest of the associated Y structure at each node. This process can be accomplished at construction time with constant overhead.

Theorem 8 The fractional cascading VO for a 2D range query q with T satisfying points is of size $O(\log N + T)$ and can be computed and verified in time $O(\log N + T)$. If the *client* accepts the $\mathcal{VO}$ V then either V was the correct $\mathcal{VO}$ or was forged.

Proof: We have already argued that the fractional cascading structure can be modeled by the search DAG and procedure we described. Therefore our security theorem applies, and we know that the verification scheme based on the search DAG model of fractional cascading is secure.

To see that the efficiency theorem applies, we note that the fractional cascading search procedure is almost identical to the procedure used for the model. So we know that it runs in $O(\log N + T)$ and thus visits $O(\log N + T)$ nodes of the DAG. It is also easy to verify that the nodes have degree at most three. The associated data at a node is either a split value for the X-tree nodes or a data value at the nodes for the Y_S arrays. Processing either value clearly takes the procedure constant time, and therefore the search DAG has bounded degree and associated data size, with $O(1)$ processing time at each node. Finally, no preprocessing of the query by P is needed, so our efficiency theorem applies to give us $\mathcal{VO}$s of size $O(\log N + T)$ and the same computational time to produce and verify a $\mathcal{VO}$. □

5.3 3-Sided Range Queries

In [1], Arge et.al. present an I/O and space efficient indexing scheme for *3-sided range queries* where for an x-range (x_l, x_u) and a lower y-limit y_l, all points (x, y) with $x_l \leq x \leq x_u$ and $y \geq y_l$ are to be selected.

On a set of N data points, their scheme uses linear storage and $O(\log_B N + t)$ I/O operations where B is the disk block size, $t = T/B$ and T is the total number of reported points. We construct a $\mathcal{VO}$ of size $O(\log N + T + B)$ for 3-sided queries based on their indexing scheme.We start with an overview of their

scheme. They divide the N points into sets of size B^2 according to their x and y coordinates. Each set of B^2 points is stored in $\Theta(B)$ *data* blocks and can be indexed using $O(B)$ values, so with constant I/O operations. We call their structure for storing B^2 points an ASV structure.

They use a main tree which we will call the X-tree, to determine which of the B^2 size sets need to be examined. The X-tree is a balanced tree with branching factor $O(B)$. Each node v in the X-tree has $O(B^2)$ points stored in an associated ASV structure, denoted Q_v. An internal node also has $O(B)$ key values which direct the search in the X-tree.

Arge et.al. show that an initial search in the X-tree can identify all nodes v whose associated ASV structure might contain answer points. This search uses $O(\log_B N + t)$ I/O operations and examines at most that many nodes in the X-tree.

Applying the Model

Since the focus of this indexing scheme is efficient I/O operations, we would like to have a verification result stated in terms of I/Os as well. If the data and hash values associated with each node in our Search DAG can be accessed with O(1) I/O's, our efficiency theorem will apply and the construction time will be in terms of I/O operations.

We now describe the Search DAG for achieving I/O efficient $\mathcal{VO}$s. Each node v in the X-tree is a node of the Search DAG, and the data associated with v is the $O(B)$ index values used to determine which of its $O(B)$ children to search. The successors of v are its children in the X-tree and a single node whose data is the index information of the associated ASV structure Q_v.

The search DAG for each ASV structure will have a node containing the $O(B)$ index values and also a node for each of the $O(B)$ data blocks whose associated data is their $O(B)$ data points. Thus, the sink nodes contain the data blocks.

We search the X-tree just as for the original data structure. For each ASV structure examined, we go to the index node and from its data determine which of its successors has the answer data.

Theorem 9 $\mathcal{VO}$s exist for 3-sided queries which can be constructed with $O(\log_B N + t)$ I/Os.

proof: Each node visited in the Search DAG corresponds directly to an I/O operation in the structure. Since each node has $O(B)$ data and successors, the $\mathcal{VO}$ data for each node can be stored in O(1) blocks, and thus can be constructed and verified using the same number of I/O's as nodes visited. □

A More Efficient Search and $\mathcal{VO}$

Although we have shown that the 3-sided query structure allows for I/O efficient $\mathcal{VO}$s, we have not discussed the precise size of the $\mathcal{VO}$ and verification time. Since each node has $O(B)$ data and successors, the $\mathcal{VO}$ size and construction time can be $O(B \log_B N + Bt) = O(B \log_B N + T)$ We can reduce the potential size and computational time of the $\mathcal{VO}$ to $O(\log N + T + B)$ by using properties of the ASV structure to carefully organize the search to reduce the time while maintaining I/O efficiency.

To avoid describing too many details of their complicated scheme, we outline our improved Search DAG and abstract the main properties of their approach which make the analysis work. First note that any time we get $\Theta(B)$ answer points from a visited node we are OK: we put $\Theta(B)$ values into the $\mathcal{VO}$, but we get $\Theta(B)$ answer points. Arge et.al. show that you get $\Theta(B)$ amortized answer points for each node visited, except for what we call *fringe nodes*. These are nodes in the X-tree along the extreme right and and left path which only partially overlap the X-range. Our improved Search DAG makes these accesses more efficient.

One part is farily easy. Since the X-tree is basically a multi-way tree keyed on the X-coordinate, we can convert each X-tree node into a binary tree of height $O(\log_2 B)$ just as we did for B-trees. The "binary" X-

tree now has constant degree and the sequence of fringe nodes in it has length $O(\log_2 B) \times \log_B N = \log_2 N$. Also, any time all the data points in a subtree rooted at an X-tree node v have Y-value below y_l, we stop exploring that subtree.

Reducing the degree of the nodes for the Q_v structures requires more detailed knowledge of their indexing scheme. Each block of data points in an ASV structure has an associated X-range x_{min}, x_{max} and Y-range y_{min}, y_{max}. A query of the form: $x_l \leq x \leq x_u$ and $y \geq y_l$, will access this block of points iff $y_{min} \leq y_l < y_{max}$ and the X-range overlaps (x_{min}, x_{max}).

To speed up searching the ASV index we use three levels of binary search trees (BST). The first level BST has the y_{min} values associated with its leaves. Associated with each such leaf is the set of blocks which satisfy the Y search constraint. The second level *Sequence* BSTs, one for each set of blocks, have the x_{min} values associated with their leaves. They are used to search for the blocks, which also overlap the query x-range. The third level BSTs, one for each block of points, are used to search for the points within a block which are ordered by x-coordinate. Note that although there are logically $O(B^2)$ of these final trees, there are only $O(B)$ blocks of points in an ASV structure, so only this many distinct trees.

Thus when we access an ASV structure we first search in the top tree to find the largest $y_{min} \leq y_l$, this gives us the root of the correct *Sequence* tree and we do a 1D range search in that tree to find blocks which overlap our query. Finally we get the actual points from each data block with another X-range search.

Theorem 10 We can answer a 3-sided query using $\Theta(\log_B N + t)$ I/O's and build a $\mathcal{VO}$ of size $\Theta(\log N + T + B)$ using linear size data structures.

Proof Sketch:

Since we can pack the binary trees associated with each X-tree or ASV structure into O(1) disk blocks, we can simulate the same search used by Arge et.al. with the same I/O performance.

For $\mathcal{VO}$ size we only need to analyze the fringe nodes. As indicated above, we now use only an $O(\log N)$ size $\mathcal{VO}$ to describe the fringe paths. When we access the associated ASV structure for a fringe node, our search trees give us the same type of performance as for a 1D range query: $O(\log_2 B + k)$ size $\mathcal{VO}$s where k is the number of points satisfying the X-range. Since there are $O(\log_B N)$ such "fringe" structures accessed, we get $O(\log N + T)$ total size. The one final point is to deal with a fringe node at the very bottom of the search which has points which are not above y_l. There are at most two such AVS structures accessed (one for the left/right path) and each may contribute $\Theta(B)$ points to the final $\mathcal{VO}$. □

6 Conclusions and Future Work

In this paper, we presented efficient methods to compute compact, secure $\mathcal{VO}$'s for a fairly broad class of data structures. The Search DAG model seems general enough to allow efficient $\mathcal{VO}$s to be computed for most data structures. One might hope however, for an even more general method which would take logical constraints on the data-structure and produce a search procedure and digest scheme. The main efficient data access scheme we can't support is hashing. Hashing can be thought of as a search with very high initial branching factor, thus it does not fit well with our model.

One major issue which needs further work is updating the database. Our data publication schemes assume fairly static data sets, an assumption which is reasonable for a wide variety of data publication scenarios. Most of the data structures we discuss do admit efficient algorithms for updates. The conversion of a data structure to a Search DAG will often leave the update properties of the original data structure intact: it will be as easy to update the Search DAG and digest as the original data structure. This was true for most of our structures, but for fractional cascading the Search DAG is harder to update (due to its chain of hash values). Similarly, the obvious search DAG for skip lists is hard to update.

An interesting future area is better higher level protocols for updates to authentic publication data sets. Our hope is to develop methods to efficiently allow frequent updates to the data sets.

Finally, authentic publication covers a broad range of secure query processing on the internet—much work remains to handle other indexing structures, types of queries, other data models, and other trust models such as data integration from different owners.

References

[1] L. Arge, V. Samoladas, and J.S. Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing In *Proc. of the 18th Symposium on Principles of Database Systems*, 346-357, 1999.

[2] M. D. Berg , M. V. Kreveld, M. Overmars and O. Schwarzkopf. *Computational Geometry.* Springer, New York, 2000.

[3] M. Goodrich , R. Tamassia Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing Preprint, 2001

[4] M. Goodrich , R. Tamassia, and A. Schwerin Implementations of an Authenticated Dictionary with Skip Lists and Commutative Hashing To appear in *DISCEX II, 2001*

[5] B. Chazelle, L.J. Guibas. Fractional Cascading: I. A Data Structuring Technique. *A*lgorithmica 1(2): 133–162, 1986.

[6] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic Third-party Data Publication, 14th IFIP 11.3 Working Conference in Database Security (DBSec 2000), 2000.

[7] S. Haber and W. S. Stornetta. How to timestamp a digital document *J. of Cryptology*, 3(2), 1991.

[8] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for Data Models with Constraints and Classes. Journal of Computer and System Sciences, 52(3), pp. 589–612, 1996.

[9] R.C. Merkle. A certified digital signature. In *Advances in Cryptology–Crypto '89*, Lecture Notes in Computer Science, Vol. 435, 218–238, Springer, 1990.

[10] M. Naor, K. Nissim. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[11] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106-119, 1997.

[12] W. Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees *C*ACM 33(6): 668–676, 1990.

[13] S. Charanjit and M. Yung. Paytree: Amortiozed Signature for flexible micropayments *Second Usenix Workshop on Electronic Commerce Proceedings*, 1996

[14] J. D. Tygar Open Problems In Electronic Commerce In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 101, 1999.